Adjoints by Automatic Differentiation

Laurent Hascoët

Adjoints by Automatic Differentiation

L. Hascoët
INRIA

Contents

0.1 Introduction

Computing accurate derivatives of a numerical model is a crucial task in many domains of Scientific Computing, and in particular for gradient-based minimization. We present Automatic Differentiation[1] (AD), a software technique to obtain derivatives of functions provided as programs (Griewank and Walther, 2008; Corliss *et al.*, 2001; Bücker *et al.*, 2005; Bischof *et al.*, 2008). Given a numerical model $F : I\!R^n \to I\!R^m$ implemented as a program `P`, AD adapts or transforms `P` into a new program that computes derivatives of F.

In the context of this summer school on Data Assimilation, the interest of Automatic Differentiation lies mainly with its so-called adjoint mode, that computes gradients efficiently. These notes focus on the aspects of AD related to the adjoint mode. In particular, an efficient adjoint mode will probably require AD by source program transformation. These notes intend to remain at the level of the principles of adjoint AD and of the software techniques that make it efficient. They are not intended to advocate one particular AD tool, although the AD tool that we are developing (Tapenade) is used at places for illustration.

Section 0.2 presents the principles of AD leading to its adjoint mode. Section 0.3 briefly presents the mathematical adjoint approach to compute gradients, underlining similarities and differences with the AD adjoint approach. It presents two application cases in Oceanography. Section 0.4 focuses on the software techniques from compiler theory (Aho *et al.*, 1986) that modern AD tools use to produce better adjoint code. Section 0.5 presents and contrasts the existing AD tools that are most likely to be useful for Data Assimilation.

0.2 Elements of AD

Given a computer algorithm `P` (identified with a piece of program) that implements a function $F : X \in I\!R^n \mapsto Y \in I\!R^m$, AD builds a new algorithm (a program piece) `P'` that computes derivatives of F by computing the analytical derivative of each elementary mathematical operation in `P`. The fundamental observation is that any run-time trace of the algorithm `P`

$$\{I_1; I_2; \dots I_p; \}$$

computes the composition of elementary mathematical functions, one per instruction I_k,

$$f_p \circ f_{p-1} \circ \cdots \circ f_1 \ \ ,$$

which we can identify to F. This is of course assuming that `P` is a correct implementation of F, i.e. the discretization and approximation employed in `P` are sufficiently accurate and do not introduce non-differentiability.

Let us clarify the correspondence between the mathematical variables $(X, Y \dots)$ and the program variables found in `P`. As imperative programs classically overwrite

[1] A warning for French-speaking readers. In these notes:

AD = **A**utomatic **D**ifferentiation $\neq$ **A**ssimilation de **D**onnées

DA = **D**ata **A**ssimilation $\neq$ **D**ifférentiation **A**utomatique

their variables to save memory space, let us call `V` the collection of all the program variables of `P` and consider that each instruction I_k (partly) overwrites `V`. With these conventions (this run-time trace of) `P` is indeed the program:

original program `P`	
	Initialize `V` *with* X
(I_1)	`V :=` f_1`(V)`
	...
(I_k)	`V :=` f_k`(V)`
	...
(I_p)	`V :=` f_p`(V)`
	Retrieve Y *from* `V`

At any given location in `P`, the program variables `V` correspond to one particular set, or vector, of mathematical variables. We will call this vector X_k for the location between instructions I_k and I_{k+1}. The set `V` is actually large enough to accomodate X, Y, or each successive X_k. At each location, `V` may thus "contain" more than the X_k but only the X_k play a role in the semantics of the program. The program instruction `V :=` f_k`(V)` actually means taking from `V` the mathematical variables X_{k-1} before the instruction and applying $f_k(X_{k-1})$ to obtain X_k. After I_k, `V` corresponds to X_k. The *Initialize with* and *Retrieve from* instructions in the program sketch define $X_0 = X$ and identify Y to X_p.

Since we identify F with a composition of functions, the chain rule of calculus gives the first-order full derivative, i.e. the Jacobian:

$$F'(X) = f'_p(X_{p-1}) \times f'_{p-1}(X_{p-2}) \times \ldots \times f'_1(X_0) \ .$$

It is thus possible in theory to adapt algorithm `P` so that it computes $F'(X)$ in addition to $F(X)$. This can be done simply by extending instruction I_1 that computes $X_1 = f_1(X_0)$ with a piece of code that computes $J_1 = f'_1(X_0) \times Id$, and by extending likewise every instruction I_k by a piece of code that computes $J_k = f'_k(X_{k-1}) \times J_{k-1}$. This transformation is local to each instruction I_k. It is not limited to straight-line code and can be applied to any program `P` with control. The extended algorithm `P'` just reproduces the control decisions taken by `P`. Of course, derivatives are valid only if the control does not change in an open neighborhood around X. Otherwise, the risk is that AD may return a derivative in cases where F is actually non-differentiable. Keeping this caveat in mind, the adapted algorithm can return J_p, the complete Jacobian $F'(X)$. However, the J_k are matrices whose height and width are both of the order of the number of variables in the original `P`, and may require too much memory space.

To work around this difficulty, we observe that the derivative object that is needed for the target application is seldom the full Jacobian matrix, but rather one of the two projections

$$F'(X) \times \dot{X} \quad \text{or} \quad \overline{Y} \times F'(X)$$

where $\dot{X}$ is some vector in $\mathbb{R}^n$ whereas $\overline{Y}$ is some row-vector in $\mathbb{R}^m$. Moreover when $F'(X)$ is needed explicitly, it is very often sparse and can therefore be retrieved from a relatively small number of the above projections. This motivates the so-called tangent and adjoint modes of AD:

- **Tangent mode:** evaluate $\dot{Y} = F'(X) \times \dot{X}$, the directional derivative of F along direction $\dot{X}$. It expands as

$$\dot{Y} = f'_p(X_{p-1}) \times f'_{p-1}(X_{p-2}) \times \ldots \times f'_1(X_0) \times \dot{X} \ . \tag{0.1}$$

  Since $\dot{X}$ is a vector, this formula is most efficiently evaluated from right to left i.e., using mathematical variables:

$$\begin{aligned}
X_0 &= X \\
\dot{X}_0 &= \dot{X} \\
X_1 &= f_1(X_0) \\
\dot{X}_1 &= f'_1(X_0) \times \dot{X}_0 \\
&\ldots \\
X_k &= f_k(X_{k-1}) \\
\dot{X}_k &= f'_k(X_{k-1}) \times \dot{X}_{k-1} \\
&\ldots \\
X_p &= f_p(X_{p-1}) \\
\dot{X}_p &= f'_p(X_{p-1}) \times \dot{X}_{p-1} \\
Y &= X_p \\
\dot{Y} &= \dot{X}_p
\end{aligned}$$

  An algorithm $\dot{\mathtt{P}}$ for this evaluation is relatively easy to construct, as the derivative instructions follow the order of the original instructions. Keeping the original program variables $\mathtt{V}$ to hold the successive X_k, and introducing a set of new program variables $\dot{\mathtt{V}}$ of the same size as $\mathtt{V}$ to hold the successive $\dot{X}_k$, $\dot{\mathtt{P}}$ writes:

tangent differentiated program $\dot{\mathtt{P}}$	
	Initialize $\mathtt{V}$ *with* X *and* $\dot{\mathtt{V}}$ *with* $\dot{X}$
$(\dot{I}_1)$	$\dot{\mathtt{V}}$:= $f'_1(\mathtt{V}) \times \dot{\mathtt{V}}$
(I_1)	$\mathtt{V}$:= $f_1(\mathtt{V})$
	...
$(\dot{I}_k)$	$\dot{\mathtt{V}}$:= $f'_k(\mathtt{V}) \times \dot{\mathtt{V}}$
(I_k)	$\mathtt{V}$:= $f_k(\mathtt{V})$
	...
$(\dot{I}_p)$	$\dot{\mathtt{V}}$:= $f'_p(\mathtt{V}) \times \dot{\mathtt{V}}$
(I_p)	$\mathtt{V}$:= $f_p(\mathtt{V})$
	Retrieve Y *from* $\mathtt{V}$ *and* $\dot{Y}$ *from* $\dot{\mathtt{V}}$

  Notice that each derivative statement $\dot{I}_k$ now precedes I_k, because I_k overwrites $\mathtt{V}$.

- **Adjoint mode:** evaluate $\overline{X} = \overline{Y} \times F'(X)$, the gradient of the scalar function $\overline{Y} \times F(X)$ derived from F and weights $\overline{Y}$. It expands as

$$\overline{X} = \overline{Y} \times f'_p(X_{p-1}) \times f'_{p-1}(X_{p-2}) \times \ldots \times f'_1(X_0) \ . \tag{0.2}$$

  Since $\overline{Y}$ is a (row) vector, this formula is most efficiently evaluated from left to right i.e., with mathematical variables:

$$
\begin{aligned}
X_0 &= X \\
X_1 &= f_1(X_0) \\
&\dots \\
X_k &= f_k(X_{k-1}) \\
&\dots \\
X_p &= f_p(X_{p-1}) \\
Y &= X_p \\
\overline{X}_p &= \overline{Y} \\
\overline{X}_{p-1} &= \overline{X}_p \times f'_p(X_{p-1}) \\
&\dots \\
\overline{X}_{k-1} &= \overline{X}_k \times f'_k(X_{k-1}) \\
&\dots \\
\overline{X}_0 &= \overline{X}_1 \times f'_1(X_0) \\
\overline{X} &= \overline{X}_0
\end{aligned}
$$

However, an algorithm that evaluates these formula is not immediate to construct, as the derivative instructions will follow the *inverse* order of the original instructions. Similarly to the tangent mode, we want the adjoint program to use only the original program's variables `V` plus a corresponding set of new program variables $\overline{\texttt{V}}$, of the same size as `V`, to hold the successive $\overline{X}_k$. In that case, we see that e.g. X_{k-1} contained in `V` will be overwritten by X_k and thus lost, before it is needed to evaluate $\overline{X}_k \times f'_k(X_{k-1})$. We will see later how this problem is solved, but let us keep in mind that there is a *fundamental penalty attached to the adjoint mode* that comes from the need of a data-flow (and control-flow) reversal.

Let us compare the run-time costs of the tangent and adjoint modes. Each run of the tangent differentiated algorithm $\dot{\texttt{P}}$ costs only a small multiple of the run-time of the original `P`. The ratio, that we will call R_t, varies slightly depending on the given `P`. Typical R_t ranges between 1 and 3. Using a simplified cost model that only counts the number of costly arithmetical operations (only `*`,`/`, and transcendentals), R_t is always less than 4. Similarly, for the adjoint differentiated algorithm $\overline{\texttt{P}}$, the run-time is only a small multiple of the run-time of `P`. The ratio, that we will call R_a, varies slightly depending on the given `P`. In the simplified cost model that only counts costly arithmetical computations, R_t and R_a are identical, but in practice $\overline{\texttt{P}}$ suffers from the extra penalty coming from the data-flow reversal. Typical R_a range between 5 and 10. Let us compare, with the help of figure 0.1, the costs of computing the complete Jacobian $F'(X)$, using no sparsity property, by employing either the tangent mode or the adjoint mode.

- With the tangent mode, we obtain $F'(X)$ column by column by setting $\dot{X}$ successively to each element of the Cartesian basis of the input space $\mathbb{R}^n$. The run time for the full Jacobian is thus $n \times R_t \times \text{runtime}(\texttt{P})$.
- With the adjoint mode, we obtain $F'(X)$ row by row by setting $\overline{Y}$ successively to each element of the Cartesian basis of the output space $\mathbb{R}^m$. The run time for

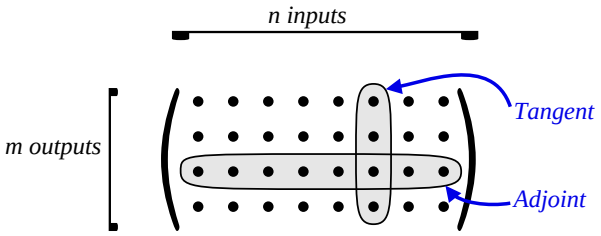

Fig. 0.1 Elements of the Jacobian computable by tangent AD and adjoint AD

the full Jacobian is thus $m \times R_a \times \text{runtime}(\mathtt{P})$.

When n is much larger than m, the adjoint mode is recommended. In particular, this is the case when gradients are needed, e.g. in optimization or in inverse problems. There are typically very few optimization criteria (often $m = 1$), and on the other hand n is often large as the optimization parameters may be functions, shapes, or other complex objects. In that case, no matter R_a being higher than R_t, the adjoint mode of AD is the only reasonable option. This is the most flagrant situation where adjoint AD can outperform all other strategies, in particular *divided differences* (i.e. evaluating $(F(X + h\dot{X}) - F(X))/h$) or even tangent AD.

Considering the design and implementation of AD tools, there are two principal ways to code the algorithms $\dot{\mathtt{P}}$ and $\overline{\mathtt{P}}$ namely, *operator overloading* and *program transformation*.

- **Operator Overloading:** if the language of P permits, we can replace the types of the floating-point variables with a new type that contains additional derivative information, and overload the arithmetic operations for this new type so as to propagate this derivative information along. Schematically, the AD tool boils down to a library that defines the overloaded type and arithmetic operations. This approach is both elegant and powerful. The overloaded library can be quickly redefined to compute higher-order derivatives, Taylor expansions, intervals... By nature, evaluation of the overloaded operations will follow the original order of P. This is fine for the tangent mode, but requires some acrobacy for the adjoint mode, bearing severe consequences on performance and/or loosing a part of the elegance of the approach.
- **Program Transformation:** We can instead decide to explicitly build a new source code that computes the derivatives. This implies parsing the original P and build an internal representation, and from it build the differentiated $\dot{\mathtt{P}}$ or $\overline{\mathtt{P}}$. This approach allows the tool to apply some global analysis on P, for instance data-flow, to produce more efficient differentiated code. This is very similar to a compiler, except that it produces source code. This approach is more development-intensive than Operator Overloading, which is one reason why Operator Overloading AD tools appeared earlier and are more numerous. It also explains why Program Transformation AD tools are perhaps slightly more fragile and need more effort to follow the continuous evolution of programming constructs and styles. On the other hand, the possibility of global analysis makes Program Transformation the

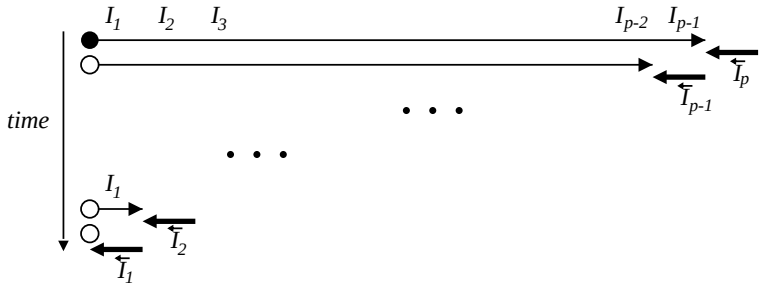

Fig. 0.2 Data-flow reversal with the Recompute-All approach. Big black dot represents storage of X_0, big white dots are retrievals

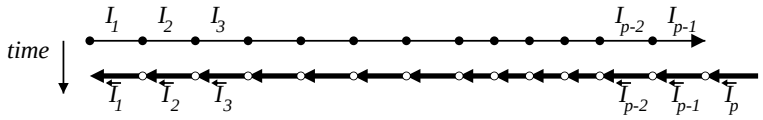

Fig. 0.3 Data-flow reversal with the Store-All approach. Small black dots represent values being recorded before overwriting. Small white dots represent corresponding restorations.

choice approach for the adjoint mode of AD, which requires control-flow and data-flow reversal and where global analysis is essential to produce efficient code.

These elements of AD are about all the background we need to describe a tangent mode AD tool. For the adjoint mode however, we need to address the question of *data-flow reversal*. We saw that the equations of the adjoint mode do not transpose immediately into a program because the values X_k are overwritten before they are needed by the derivatives. In the context of Operator Overloading, the natural stategy for the adjoint mode is to have the overloaded operations write on a unique "tape" the long log of program variables being read, combined with differentiable operations, and overwritten. In compiler terminology, this is a kind of *three-address code*. Later, this tape is read from end to beginning by a special code that computes the derivatives. This is indeed a recording not only of the data-flow, but also of every arithmetic operation done. There are variations from this scheme but the tape remains very large. In the context of Program Transformation, only the data-flow need be reversed. There are two ways to solve this problem, and a variety of combinations between them.

- *Recompute-All:* For each derivative instruction $\overleftarrow{I_k}$, we recompute the X_{k-1} that it requires by a repeated execution of the original code, for the stored initial state X_0 to instruction I_{k-1} that computes X_{k-1}. This is sketched on figure 0.2. The extra cost in memory is only the storage of X_0. On the other hand, the extra cost in run time is quadratic in p.
- *Store-All:* Each time instruction I_k overwrites a part of `V`, we record this part of `V` into a stack just before overwriting. Later, we restore these values just before executing $\overleftarrow{I_k}$ This is sketched on figure 0.3. The extra cost in memory is proportional to p, whereas the extra cost in run time comes from stack manipulation, usually

minimal and proportional to p. We sketch a program $\overline{\mathtt{P}}$ that uses the Store-All approach, using `push` and `pop` primitives for stack manipulations, and definining $\mathbf{out}(I_k)$ to be the subset of the variables `V` that are effectively overwritten by I_k. We see two successive sweeps in $\overline{\mathtt{P}}$. The *forward sweep* $\overrightarrow{\mathtt{P}}$ is essentially a copy of `P` augmented with storage of overwritten values. The *backward sweep* $\overleftarrow{\mathtt{P}}$ is the computation of the derivatives, in reverse order, augmented with retrieval of recorded values. Due to retrievals, the exit `V` does not contain the original result Y.

adjoint differentiated program $\overline{\mathtt{P}}$	**(Store-All)**
	Initialize `V` *with* X *and* $\overline{\mathtt{V}}$ *with* $\overline{Y}$
	`push(`$\mathbf{out}(I_1)$`)`
(I_1)	`V :=` f_1`(V)`
	...
	`push(`$\mathbf{out}(I_k)$`)`
(I_k)	`V :=` f_k`(V)`
	...
	`push(`$\mathbf{out}(I_{p-1})$`)`
(I_{p-1})	`V :=` f_{p-1}`(V)`
$(\overleftarrow{I_p})$	$\overline{\mathtt{V}}$ `:=` $\overline{\mathtt{V}} \times f'_p$`(V)`
	`pop(`$\mathbf{out}(I_{p-1})$`)`
	...
	`pop(`$\mathbf{out}(I_k)$`)`
$(\overleftarrow{I_k})$	$\overline{\mathtt{V}}$ `:=` $\overline{\mathtt{V}} \times f'_k$`(V)`
	...
	`pop(`$\mathbf{out}(I_1)$`)`
$(\overleftarrow{I_1})$	$\overline{\mathtt{V}}$ `:=` $\overline{\mathtt{V}} \times f'_1$`(V)`
	Retrieve $\overline{X}$ *from* $\dot{\mathtt{V}}$

These data-flow reversal strategies can be adapted to programs `P` that are no longer straight-line, but that use control. This implies that the control decisions of the original code (branch taken, number of iterations, ...) or equivalently of the forward sweep, must be made available in reverse for the backward sweep. In a Store-All context, this can be done just like for data: control-flow decisions must be recorded at the exit of the control structure (conditional, loop,...) and then retrieved in the backward sweep to control the backward execution. This can use the same stack as data-flow reversal. The conjunction of the recorded control and data values is called the *trajectory*.

In practice, neither pure Recompute-All nor pure Store-All can be applied to large programs, because of their respective cost in run-time or memory space. This problem occurs also with overloading-based AD, which behaves like Store-All in this respect. Trade-offs are needed, and the classical trade-off is called *checkpointing*.

- In the Recompute-All approach, checkpointing means choosing a part `C` of `P` and storing the state upon exit from this part. Recomputing can then start from this state instead of the initial state. This is sketched on the top-left part of figure 0.4. At the cost of storing one extra state, the run-time penalty has been divided roughly by two. Checkpoints can be nested to further reduce the run-time penatly, as shown on the bottom-left of figure 0.4.
- In the Store-All approach, checkpointing means choosing a part `C` of `P` and *not* recording the overwritten values during `C`. Before the backward sweep reaches $\overleftarrow{\mathtt{C}}$, `C` is run again from a stored state this time with recording. This is sketched on the

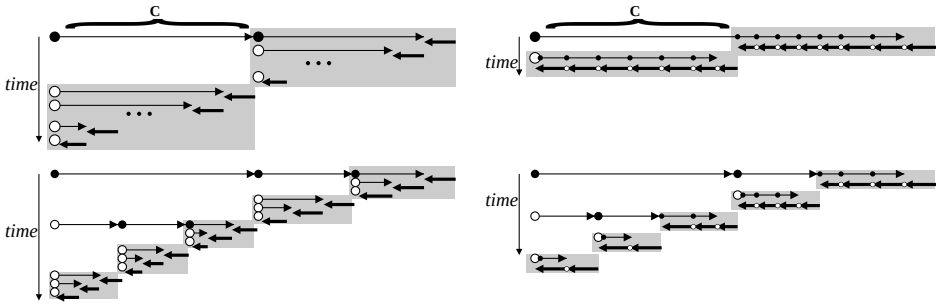

Fig. 0.4 Checkpointing with the Recompute-All (*left*) and Store-All (*right*) approaches. The shaded areas reproduce the basic pattern of the chosen approach. *Top:* single checkpointing, *bottom* nested checkpointing.

top-right part of figure 0.4. At the cost of storing one extra state and of running `C` twice, the peak memory used to record overwritten data is divided roughly by two. Checkpoints can be nested to further reduce the peak memory consumption, as shown on the bottom-right of figure 0.4.

Notice on figure 0.4 that the execution scheme at the bottom, for nested checkpoints, become very similar. Recompute-All and Store-All are the two ends of a spectrum, with optimal checkpointing scheme(s) lying somewhere in between. A good placement of (nested) checkpoints is crucial for efficient adjoint differentiation of large codes. Natural candidates to become a checkpointed part are procedure calls and loop bodies, but any piece of code with a single entry point and a single exit point can be chosen. There is no formula nor efficient algorithm to find this optimal placement of checkpoints, except in the case (Griewank, 1992) of a loop with a known number of iterations all of the same cost. A good enough default strategy is to apply checkpointing at the level of each procedure call. In practice, it is important to give the end-user the freedom to place checkpoints by hand. Good placements of checkpoints perform similarly: their memory and run-time costs grow with the logarithm of the run-time of `P` or more precisely:

- the peak memory size during execution, to store states and record data, grows like the logarithm of the run-time of `P`.
- the maximum number of times a checkpointed piece of the program is re-executed, which approximates the slowdown factor of $\overline{\mathtt{P}}$ compared to `P`, also grows like the logarithm of the run-time of `P`. This explains why the slowdown ratio R_a of $\overline{\mathtt{P}}$ compared to `P` can be larger by a few units than the ratio R_t of $\dot{\mathtt{P}}$ compared to `P`.

Actually, we do not need to store the entire memory state to checkpoint a program piece `C`. What must be stored is called the *snapshot*. In a Store-All context, we can see that a variable need not be in the snapshot if it is not used by `C`. Likewise, a variable need not be in the snapshot if it is not overwritten between the initial execution of `C` and the execution of its adjoint $\overline{\mathtt{C}}$.

0.3 Application of adjoint AD to Data Assimilation

Assume we have a physical model that represents the way some unknown values determine some measurable values. When this model is complex, its inverse problem is nontrivial. From this physical model we get a mathematical model, which is in general a set of partial differential equations. Let us also assume that by discretization and resolution of the mathematical model, we get a program that computes the measurable values from the unknown values.

Using the formalism of *optimal control theory* (le Dimet and Talagrand, 1986), we are studying the state W of some system, W is defined for every point in space, and also – if time is involved – for every instant in an observation period $[0, T]$. The mathematical model relates the state W to a number of external parameters, which are the collection of initial conditions, boundary conditions, model parameters, etc, i.e. all the values that determine the state. Some of these parameters, that we call γ, are the unknown of our inverse problem. This relation between W and γ is implicit in general. It is a set of partial differential equations that we write:

$$\Psi(\gamma, W) = 0 \tag{0.3}$$

Equation (0.3) takes into account all external parameters, but we are only concerned here by the dependence on γ. In optimal control theory, we would call γ our control variable. Any value of γ thus determines a state $W(\gamma)$. We can extract from this state the measurable values, and of course there is very little chance that these values exactly match the values actually measured W_{obs}. Therefore we start an optimization cycle to modify the unknown values γ, until the resulting measurable values match best. We thus define a *cost function* that measures the discrepancy on the measurable values in $W(\gamma)$. In practice, not all values in $W(\gamma)$ can be measured in W_{obs}, but nevertheless we can define this cost function J as the sum at each instant of some squared norm of the discrepancy of each measured value $\|W(\gamma) - W_{obs}\|^2$.

$$j(\gamma) = J(W(\gamma)) = \frac{1}{2} \int_{t=0}^{T} \|W(\gamma)(t) - W_{obs}(t)\|^2 dt \tag{0.4}$$

Therefore the inverse problem is to find the value of γ that minimizes $j(\gamma)$, i.e. such that $j'(\gamma) = 0$. If we use a gradient descent algorithm to find γ, we need to find $j'(\gamma)$ for each γ. To this end, the mathematical approach first applies the chain rule to equation (0.4), yielding:

$$j'(\gamma) = \frac{\partial J(W(\gamma))}{\partial \gamma} = \frac{\partial J}{\partial W} \frac{\partial W}{\partial \gamma} \tag{0.5}$$

The derivative of W with respect to γ comes from the state implicit equation (0.3), which we differentiate with respect to γ to get:

$$\frac{\partial \Psi}{\partial \gamma} + \frac{\partial \Psi}{\partial W} \frac{\partial W}{\partial \gamma} = 0 \tag{0.6}$$

Assuming this can be solved for $\frac{\partial W}{\partial \gamma}$, we can then substitute it into equation (0.5) to get:

$$j'(\gamma) = -\frac{\partial J}{\partial W}\frac{\partial \Psi}{\partial W}^{-1}\frac{\partial \Psi}{\partial \gamma} \tag{0.7}$$

Now is the time to take complexity into account. Equation (0.7) involves one system resolution and then one product. Nowadays both Ψ and W are discretized with millions of dimensions. $\frac{\partial \Psi}{\partial W}$ is definitely too large to be computed explicitly, and therefore its inverse cannot be computed nor stored either. We notice that $\frac{\partial \Psi}{\partial \gamma}$ has many columns, whereas $\frac{\partial J}{\partial W}$ has only one row. Therefore the most efficient way to compute the gradient $j'(\gamma)$ is the *adjoint method*: first compute $\frac{\partial J}{\partial W}$, then run an iterative resolution for

$$\frac{\partial J}{\partial W}\frac{\partial \Psi}{\partial W}^{-1} \tag{0.8}$$

and then multiply the result (called the *adjoint state* Π) by $\frac{\partial \Psi}{\partial \gamma}$.

Recall now that we already have a resolution program, i.e. a procedure $\mathtt{P}_\Psi$ which, given γ, returns W_γ, and a procedure $\mathtt{P}_j$ which, given a W, evaluates the cost function, i.e. the discrepancy between W and the observed W_{obs}. Adjoint AD of the program that computes

$$j = \mathtt{P}_j(\mathtt{P}_\Psi(\gamma))$$

directly gives the gradient of j, i.e. the desired $j'(\gamma)$. This is indeed very close to the mathematical resolution with the adjoint state. In both cases, $\frac{\partial J}{\partial W}$ is computed first, thus guaranteeing that no large square matrix is stored. Things differ a little for the following stage, because a program is by essence explicit and therefore the resolution for Π and multiplication with $\frac{\partial \Psi}{\partial \gamma}$ are done jointly. Apart from that, adjoint AD can really be considered as the discrete equivalent (on programs) of the above adjoint method.

As a first application, we considered the oceanography code OPA 9.0 (Madec *et al.*, 1998) on a simple configuration known as GYRE. It simulates the behavior of a rectangular basin of water put on the tropics between latitudes 15^o and 30^o, with the wind blowing to the East. Our control variables γ are the temperature field in the complete domain, and our cost function $j(\gamma)$ is the discrepancy with respect to measurements of the heat flux across some boundary at the northern angle 20 days later. Figure 0.5 shows one gradient $j'(\gamma)$ computed by adjoint AD. This system is discretized with $32 \times 22 \times 31$ nodes and 4320 time steps. The original simulation takes 26 seconds, and the differentiated program computes the gradient above in 205 seconds, which is only 7.9 times as long. Of course checkpointing is absolutely necessary to reverse this long simulation, yielding several recomputations of the same program steps, but nevertheless the factor 7.9 is much better than what tangent AD would require. Checkpointing and storage of the trajectory use a stack that reaches a peak size of 481 Mbytes.

In a second, larger application we considered the NEMO configuration of OPA (2^o grid cells, one year simulation) for the north Atlantic. Figure 0.6 shows the gradient of the discrepancy to measurements of the heat flux across the 29^{th} parallel, with respect to the temperature field one year before. With this discretization, there are 9100 control parameters in γ. The adjoint code that computes the gradient with respect to these 9100 parameters takes 6.5 times as long as the original code. The size of the

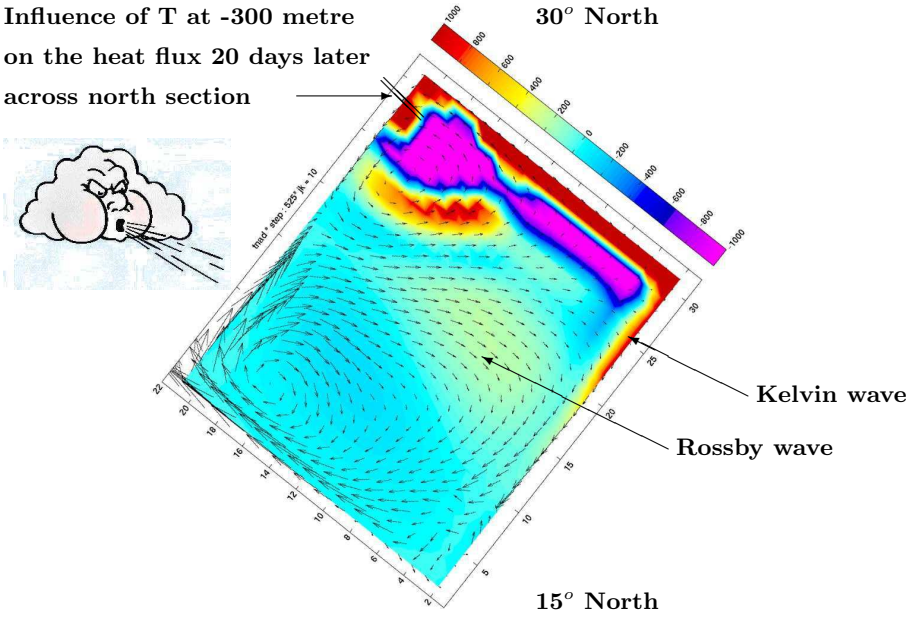

Fig. 0.5 *Oceanography gradient by adjoint AD on OPA Gyre*

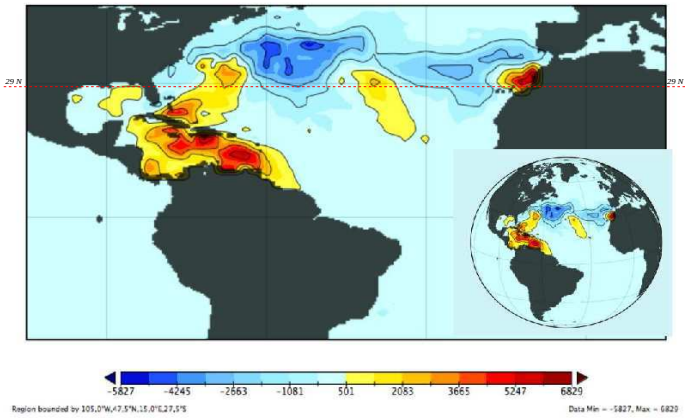

Fig. 0.6 *Oceanography gradient by adjoint AD on OPA Nemo*

stack that stores the trajectory and the snapshots for checkpointing reaches a peak of 1591 Mbytes.

0.4 Improving the adjoint AD code

AD tools will produce AD adjoints automatically, following the principles described in section 0.2. The resulting adjoint code can be easily criticized: a skilled programmer can always write a more efficient code, at least in theory, using long and careful work.

Nevertheless, AD tool developers constantly observe what skilled adjoint programmers do, and try to generalize the improvements, model them, and incorporate them into the AD tools. In other situations, the AD tool can be extended with options to let the end-user request a predefined improvement at a given location. We will illustrate here only a few such improvements.

Activity analysis is systematic in many Program Transformation AD tools. It is useful for both tangent and adjoint AD. Assume that the user has specified the set of *independents*, i.e. the input variables with respect to which differentiation must be done, and the set of *dependents*, i.e. the output variables that must be differentiated. Activity analysis propagates forward the *varied* variables, i.e. those that depend on some independent in a differentiable way. It also propagates backwards the *useful* variables, i.e. those that influence some dependent in a differentiable way. At each given location in the code, the derivative variable of an original variable needs to be mentioned *only* if this original variable is both varied and useful. In all other situations, the derivative is either certainly null or certainly useless, and therefore need not appear in the differentiated code. Activity analysis is just one of the many data-flow analyses that AD tools use, applying techniques from compiler theory. Figure 0.7 illustrates

original program	**tangent AD**	**adjoint AD**

`x = 1.0`
`z = x*y`
`t = y**2`
`IF (t .GT. 100) ...`
	`x = 1.0`
	`zd = x*yd`
	`z = x*y`
	`t = y**2`
	`IF (t .GT. 100) ...`
		`x = 1.0`
		`z = x*y`
		`t = y**2`
		`IF (t .GT. 100) ...`
		`...`
		`yb = yb + x*zb`

Fig. 0.7 *Instructions simplifications due to Activity Analysis*

the benefits of activity analysis. `x` immediately becomes not varied, and `t` is useless. Therefore, the AD tool knows that `xd` and `tb` are null and can be simplified. In some situations, we can even choose not to set them explicitly to zero.

TBR analysis is systematic, and specific to Store-All adjoint AD. The Store-All approach says that the value that is being overwritten by an assignment must be (1) stored just before this assignment in the forward sweep and (2) restored before the adjoint of this assignment in the backward sweep. However, TBR analysis can detect that a particular value is actually not used in the derivative computations. To put it short, this is the case for every value that is used only in *linear* computations. It is not necessary to store and restore these values, and this saves a significant amount of trajectory in the stack memory. In the example of figure 0.8, TBR analysis could prove that neither `x` nor `y` were needed by the differentiated instructions, and therefore these variables need not be `PUSH`'ed on nor `POP`'ed from the stack.

original program	**adjoint mode:** **naïve backward sweep**	**adjoint mode:** **backward sweep TBR**

`x = x + EXP(a)`
`y = x + a**2`
`a = 3*z`
	`CALL POPREAL4(a)`
	`zb = zb + 3*ab`
	`ab = 0.0`
	`CALL POPREAL4(y)`
	`ab = ab + 2*a*yb`
	`xb = xb + yb`
	`yb = 0.0`
	`CALL POPREAL4(x)`
	`ab = ab + EXP(a)*xb`
		`CALL POPREAL4(a)`
		`zb = zb + 3*ab`
		`ab = 0.0`
		
		`ab = ab + 2*a*yb`
		`xb = xb + yb`
		`yb = 0.0`
		
		`ab = ab + EXP(a)*xb`

Fig. 0.8 *Removing unnecessary storage through TBR analysis*

Binomial checkpointing is the optimal way to organize checkpointing for a special class of loops that essentially correspond to time-stepping iterations (Griewank and Walther, 2008). See figure 0.9. This is a crucial improvement to adjoint AD of

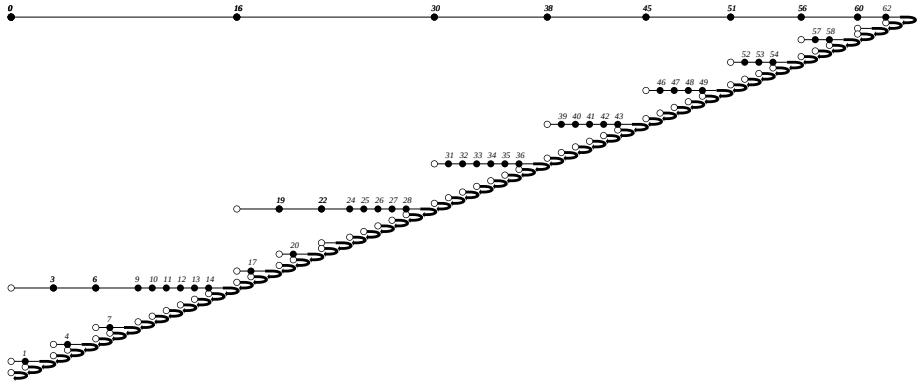

Fig. 0.9 *Optimal binomial checkpointing on 64 time steps*

most unsteady simulation codes. It is more powerful than multi-level checkpointing. Basically, binomial checkpointing recursively divides sequences of time steps into two sub-sequences of unequal length given by a binomial law. The peak memory required by the tape, as well as the slowdown factor coming from recomputations, only grow like the logarithm of the number of time steps. Binomial checkpointing is somewhat intricate to implement in general, so it must be provided by the AD tool. However the tool cannot discover the time-stepping loops, and the end-user must designate them with a directive to the AD tool.

Linear solvers are better differentiated by hand. Both in tangent and adjoint modes, there are simple formulae that propagate derivatives through them very efficiently. In contrast, standard AD will probably do a very poor job differentiating all the minute irrelevant details that are found in an optimized linear solver routine. Here also, there is no hope that an AD tool can detect automatically that a subroutine is

a linear solver. This is probably undecidable anyway. So the AD tool must be guided by user directives. The AD tool will then either apply some predefined differentiation strategy to the linear solver, or just leave a "hole" in the adjoint code and ask the user to fill the hole with the very short differentiated subroutine for the linear solver. This last approach is often referred to as the *black-box* approach and is not limited to linear solvers: it applies to any procedure for which there is a clever differentiation, or to external routines for which the source code is unavailable.

0.5 AD tools

Here is our (partial) view of available AD tools and of how one might classify them. The best source is the `www.autodiff.org` site, which is the portal managed by our colleagues in Aachen and Argonne in the name of the AD community. It turns out that plenty of experiments were made that adapt the concepts of AD to particular environments. There was for instance a clever introduction of tangent AD into Microsoft's Excel. However in the present context, we will focus on the tools that we feel can be applied to inverse problems and data assimilation of industrial size.

As we saw, some AD tools rely on program overloading rather than program transformation. In general this makes the tool easier to implement. However some overloading-based AD tools can become very sophisticated and efficient, and represent a fair bit of hard work too. Overloading-based AD tools exist only for target languages that permit some form of overloading, e.g. C++ and Fortran95. Overloading-based AD tools are particularly adapted for differentiations that are mostly local to each statement, i.e. no fancy control flow rescheduling is allowed. On the other hand, these local operations can be very sophisticated, more than what transformation-based AD tools generally provide. For instance, overloading-based AD tools can generally compute not only first, but also second, third derivatives and so on, as well as Taylor expansions or interval arithmetic. **Adol-C** (Walther and Griewank, 2012), is an excellent example of overloading-based AD tool. **FADBAD/TADIFF** are other examples.

The AD tools based on program transformation parse and analyze the original program and generate a new source program. These tools share their general architecture: a front-end very much like a compiler, followed by an analysis component, a differentiation component, and finally a back-end that regenerates the differentiated source. They differ in particular in the language that they recognize and differentiate, and in the AD modes that they provide. They also exhibit some differences in AD strategies mostly about the adjoint mode. The best known other transformation-based AD tools are the following:

- Tapenade (Hascoët and Pascual, 2004) provides tangent and adjoint differentiation of Fortran (77 and 95) and C. Adjoint mode uses the Store-All approach to restore intermediate values. This Store-All approach is selectively replaced by recomputation in simple appropriate situations. Checkpointing is applied by default at the level of procedure calls, but can be triggered or deactivated at other places through used directives.
- Adifor (Carle and Fagan, 2000) differentiates Fortran77 codes in tangent mode. Adifor once was extended towards the adjoint mode (Adjfor), but we believe this know-how has now been re-injected into the OpenAD framework, described below.

- Adic can be seen as the C equivalent of Adifor. However, it has been lately rebased on a completely different architecture, from the OpenAD framework. Adic differentiates ANSI C programs in tangent mode, with the possibility to obtain second derivatives.
- OpenAD/F (Utke *et al.*, 2008) differentiates Fortran codes in tangent and adjoint modes. The general framework of OpenAD claims (like Tapenade) that only front-end and back-end should depend on the particular language, whereas the analysis and differentiation part should work on a language-independent program representation. This is why OpenAD was able to host Adic. OpenAD/F is made of Adifor and Adjfor components integrated into the OpenAD framework. Its strategy to restore intermediate values in adjoint AD is extremely close to Tapenade's.
- TAMC (Giering, 1997), through its commercial offsprings TAF and TAC++ differentiates Fortran and C files. TAF also differentiates Fortran95 files, under certain restrictions. TAF is commercialized by the FastOpt company in Hamburg, Germany. Differentiation is provided in tangent and adjoint mode, with the Recompute-All approach to restore intermediate values in adjoint AD. Checkpointing and an algorithm to avoid useless recomputations (ERA) are used to avoid explosion of run-time. TAF also provides a mode that efficiently computes the sparsity pattern of Jacobian matrices, using bit-sets.

There are also AD tools that directly interface to an existing compiler. In fact, these are modifications to the compiler such that the compiler performs AD at compile time. This new AD functionnality is triggered by new constructs or new directives added into the application language. In a sense, this amounts to modifing the application language so that it has a new "differentiation" operator. For instance the NAGWare Fortran95 compiler embeds AD facilities (Naumann and Riehme, 2005), that are triggered by user directives in the Fortran source. To our knowledge, its adjoint mode uses a strategy equivalent to operator overloading.

There are AD tools that target higher-level languages, such as MATLAB. We know of ADiMat, MAD, and INTLAB. Even when they rely on operator overloading, they may embed a fair bit of program analysis to produce efficient differentiated code.

0.6 Conclusion

We have presented Automatic Differentiation, and more precisely the fundamental notions that are behind the AD tools that use source program transformation. We shall use figure 0.10 as a visual support to compare AD with other ways to obtain derivatives. Our strongest claim is that if you need derivatives of functions that are already implemented as programs, then you should seriously consider AD. At first thought, it is simpler to apply the Divided Differences method (sometimes known also as "Finite Differences"), but its inaccuracy is its major drawback.

Notice that Divided Differences sometimes behave better when the implemented function is not differentiable, because its very inaccuracy has the effect of smoothing discontinuities of the computed function. Therefore Divided Differences can be an option when one only has piecewise differentiability. Also, it is true that Divided Differences may actually cost a little less than the tangent mode, which is their AD

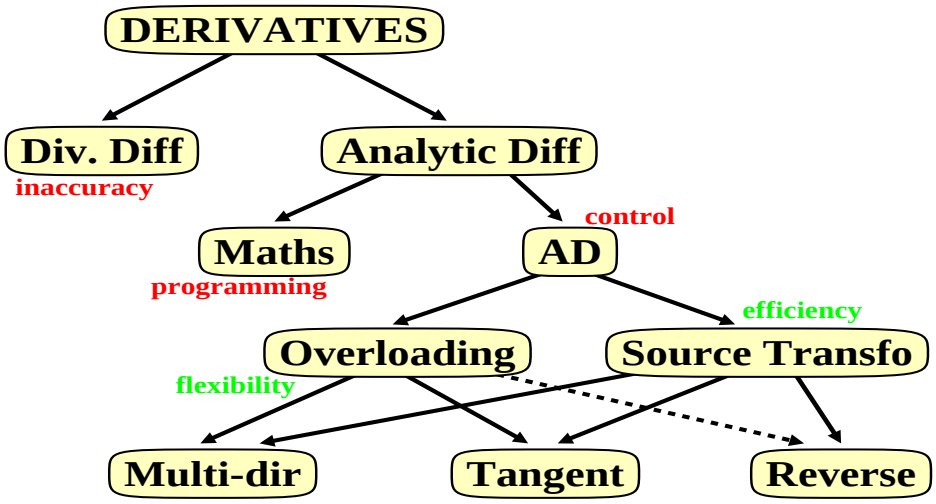

Fig. 0.10 *AD and other ways to compute derivatives*

equivalent. Nevertheless when it is possible, it is safer to look for exact analytical derivatives.

Divided Differences are definitely a poor choice if you need gradients.

Then again two options arise: one can consider the problem of finding the derivatives as a new mathematical problem, with mathematical equations (e.g. adjoint equations) that must be discretized and solved numerically. This is a satisfying mathematical approach, but one must be aware of its development cost.

AD is a very promising alternative when the function to be differentiated has already been implemented. In a sense, AD reuses the resolution strategy that has been implemented for the original function into the resolution of its derivatives. When the original model or code changes, AD can be applied again at very little cost.

There are still open questions of non-differentiability introduced by the program's control, or the fact that the iterative resolution of the derivatives is not always guaranteed to converge at the same speed as the original function resolution. But in practice, AD returns derivatives that are just as good as those returned by the "mathematical" approach above.

Inside the AD methods we distinguish Overloading-based approaches, which are more flexible and can be adapted to all sorts of derivatives and similar concepts. On the other hand, we advocate source-transformation-based tools for the well-identified goals of tangent and adjoint first-order derivatives. Source transformation gives it full power when it performs global analyses and transformations on the code being differentiated.

Source Transformation AD is really the best approach to the adjoint mode of AD, which computes gradients at a remarkably low cost. Adjoint AD is a discrete equivalent

of the adjoint methods from control theory. Adjoint AD may appear puzzling and even complex at first sight. But AD tools apply it very reliably so that a basic understanding of it generally suffices. Adjoint AD is really the choice method to get the gradients required by inverse problems (e.g. data assimilation) and optimization problems.

AD tools can build highly optimized derivative programs in a matter of minutes. AD tools are making progress steadily, but the best AD will always require end-user intervention. Moreover, one must keep in mind the limitations of AD in order to make a sensible usage of it. Fundamentally:

- real programs are always only piecewise differentiable, and only the user can tell if these algorithmic discontinuities will be harmful or not.
- iterative resolution of the derivatives may not converge as well as the original program, and the knowledge of the Numerical Scientist is invaluable to study this problem.
- adjoint AD of large codes will always require a careful profiling to find the best storage/recomputation trade-off.

There are also a number of technical limitations to AD tools, which may be partly lifted in the future, but which are the current frontier of AD tool development:

- Dynamic memory in the original program is a challenge for the memory restoration mechanism of the adjoint mode.
- Object-oriented languages pose several very practical problems, because they far more intensively use the mechanisms of overloading and dynamic allocation. Data-Flow analysis of object-oriented programs may become harder and return less useful results.
- Parallel communications or other system-related operations may introduce a degree of randomness in the control flow, which is then hard to reproduce, duplicate, or reverse.

References

Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Bischof, C., Bücker, M., Hovland, P., Naumann, U., and Utke, J. (ed.) (2008). *Advances in Automatic Differentiation*. Volume 64, Lecture Notes in Computational Science and Engineering. Springer, Berlin.

Bücker, M., Corliss, G., Hovland, P., Naumann, U., and Norris, B. (ed.) (2005). *Automatic Differentiation: Applications, Theory, and Implementations*. Volume 50, Lecture Notes in Computational Science and Engineering. Springer, New York, NY.

Carle, A. and Fagan, M. (2000). ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University.

Corliss, G., Faure, C., Griewank, A., Hascoët, L., and Naumann, U. (ed.) (2001). *Automatic Differentiation: from Simulation to Optimization*. Computer and Information Science. Springer, New York, NY.

Giering, R. (1997). Tangent linear and Adjoint Model Compiler, Users manual. Technical report. http://www.autodiff.com/tamc.

Griewank, A. (1992). Achieving logarithmic growth of temporal and spatial complexity in reverse Automatic Differentiation. *Optimization Methods and Software*, **1**, 35–54.

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (2nd edn). Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA.

Hascoët, L. and Pascual, V. (2004). TAPENADE 2.1 user's guide. Rapport technique 300, INRIA.

le Dimet, F.-X. and Talagrand, O. (1986). Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects. *Tellus*, **38A**, 97–110.

Madec, G., Delecluse, P., Imbard, M., and Levy, C. (1998). OPA8.1 ocean general circulation model reference manual. Technical report, Pole de Modelisation, IPSL.

Naumann, U. and Riehme, J. (2005). Computing adjoints with the NAGWare Fortran 95 compiler. pp. 159–169 in (Bücker, Corliss, Hovland, Naumann and Norris, 2005).

Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., and Wunsch, C. (2008). OpenAD/F: A modular, open-source tool for Automatic Differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, **34**(4), 18:1–18:36.

Walther, A. and Griewank, A. (2012). Getting started with ADOL-C. In *Combinatorial Scientific Computing* (ed. U. Naumann and O. Schenk), Chapter 7, pp. 181–202. Chapman-Hall CRC Computational Science.